Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs

René Just, Darioush Jalali, and Michael D. Ernst
Computer Science and Engineering
University of Washington
Seattle, WA, USA
{rjust, darioush, mernst}@cs.washington.edu

ABSTRACT

Empirical studies in software testing research may not be comparable, reproducible, or characteristic of practice. One reason is that real bugs are too infrequently used in software testing research. Extracting and reproducing real bugs is challenging and as a result hand-seeded faults or mutants are commonly used as a substitute.

This paper presents Defects4J, a database and extensible framework providing real bugs to enable reproducible studies in software testing research. The initial version of Defects4J contains 357 real bugs from 5 real-world open source programs. Each real bug is accompanied by a comprehensive test suite that can expose (demonstrate) that bug. Defects4J is extensible and builds on top of each program's version control system. Once a program is configured in Defects4J, new bugs can be added to the database with little or no effort.

Defects4J features a framework to easily access faulty and fixed program versions and corresponding test suites. This framework also provides a high-level interface to common tasks in software testing research, making it easy to conduct and reproduce empirical studies. Defects4J is publicly available at `http://defects4j.org`.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Testing tools

Keywords

Bug database, real bugs, testing framework

1. INTRODUCTION

Reproducibility of empirical studies in software testing research is challenging due to the lack of widely accepted and easy-to-use databases of real bugs. A large number of previously found and fixed bugs are documented in bug tracking systems of open source projects. Yet, extracting, reproducing, and isolating those real bugs requires considerable effort, and hence real bugs are rarely used in software testing research. Existing databases or repositories that provide faulty program versions [2, 4] offer only a very limited quantity of real bugs and as a consequence mutants or hand-seeded faults are commonly used as a substitute. However, mutants and hand-seeded faults differ from inadvertently introduced bugs, and thus might not be suitable for evaluating testing techniques [6].

This paper presents Defects4J, a database and framework providing 357 real bugs to support software testing research. The acronym Defects4J reflects its purpose: A **d**atabase of **e**xisting **f**aults to **e**nable **c**ontrolled **t**esting **s**tudies **for J**ava programs. In summary, Defects4J makes the following contributions:

- Defects4J implements a new approach for reproducing and isolating real bugs from version control history. In this context, isolation means that the bug fix does not include irrelevant changes such as features or refactorings. Reproducibility is ensured through an accompanying test suite that includes at least one test case that exposes the bug — that is, the test case succeeds on the fixed but fails on the faulty program version.
- Defects4J enables reproducibility in software testing research by providing a populated database of isolated real bugs for real-world programs. The initial version of Defects4J provides 357 real bugs for 5 large open source programs. We expect the database to grow because Defects4J is designed to be extensible. Defects4J builds on top of the projects' version control systems — once a program is configured in Defects4J, new bugs, e.g., obtained from newly reported bug fixes, can be added to the database with little or no effort.
- Defects4J provides a database abstraction layer that eases the use of the bug database. This abstraction provides a uniform interface for checking out faulty and fixed program versions. It also provides uniform access to build and execution targets by abstracting the build systems of all programs.
- Defects4J features a test execution framework that eases the implementation of tools for experiments in software testing research. This framework provides several components for common tasks in software testing such as test execution, test generation, and code coverage or mutation analysis.

Figure 1 visualizes the overall architecture of Defects4J. Section 2 describes how we populated the bug database and the subsequent Sections 3, 4, and 5 detail the individual parts of Defects4J.

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2628055

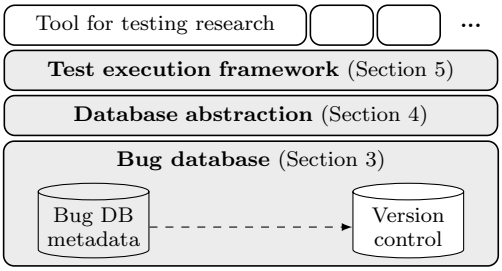

Figure 1: The big picture: architecture of Defects4J.

2. REPRODUCING AND ISOLATING BUGS FROM VERSION CONTROL HISTORY

This section describes the methodology we used to populate Defects4J's bug database. It also sheds light on challenges to collecting and reproducing real bugs from version control history and how we addressed those challenges.

The overall goal is to identify real bugs (i.e., bugs fixed by a developer), and to obtain, for each bug, a faulty (V_{bug}) and a fixed (V_{fix}) source code version that differ by only the bug fix. Specifically, each real bug included in Defects4J's bug database fulfills the following requirements:

- **The bug is related to source code**
  A developer explicitly labeled the commit of V_{fix} as a bug fixing commit, and the bug fix applies to the source code — bug fixes within the build system, configuration files, documentation, or tests are not included.
- **The bug is reproducible**
  V_{fix} is accompanied by at least one test that passes on V_{fix} but fails on V_{bug}, and the bug is reproducible using the project's build system and an up-to-date JVM[1].
- **The bug is isolated**
  The bug fix (i.e., the diff between V_{bug} and V_{fix}) does not include unrelated changes such as features or refactorings.

Defects4J includes a toolset that supports automation of populating the bug database with real bugs from version control history.

2.1 Identify Real Bugs Fixed by Developers

A fundamental challenge when collecting bugs is deciding what constitutes a bug, and what does not. During our analyses of version control and bug tracking systems, we encountered several types of fixes not related to the source code, or features, which were classified as a "bug fix" by developers. Examples of bugs unrelated to the source code include faulty or incomplete documentation, faulty tests or test input data, and faulty build system configurations.

In Defects4J, an automated step identifies candidates for inclusion. A commit is considered a fixed program version V_{fix} if 1) the commit log references a bug id of the bug tracking system or 2) the bug tracking system references a commit id of the version control system. A commit is not relevant if it does not include source code changes.

2.2 Reproduce Real Bugs

A committed source code version might fail some of its own tests. Yet, those failing tests do not necessarily expose a bug in the source code. Similarly, the existence of tests that sporadically fail might lead to non-deterministic results.

In Defects4J, an automated step removes all tests that fail on V_{fix} from the test suite before attempting to expose the bug in V_{bug} — the majority of program versions were accompanied by failing tests. The automated step then executes the test suite on V_{bug}, and a bug is considered reproducible if at least one test case fails on V_{bug} due to the fault.

[1]We reproduced all bugs in Defects4J using a Java 7 runtime environment and the OpenJDK Java virtual machine (JVM).

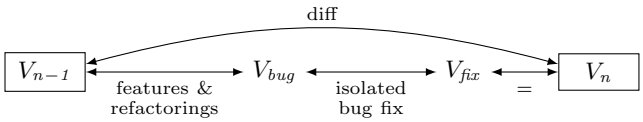

Figure 2: Source code versions V_{bug} and V_{fix} that differ by only a bug fix. V_{n-1} and V_n represent the source code versions of two consecutive revisions in a project's version control history.

2.3 Isolate Real Bugs

Isolating the bug is crucial to support testing experiments that rely on V_{bug} and V_{fix}. Consider, for example, the evaluation of a test generation approach. If the difference between V_{bug} and V_{fix} would include features or refactorings, then a test suite generated for V_{fix} could fail on V_{bug} simply because V_{bug} misses features unrelated to the bug.

We manually reviewed the source code diffs of reproducible bugs to verify that they did not include irrelevant changes — if necessary, we isolated the bug fix from the source code diff. Figure 2 visualizes the relationship between V_{bug} and V_{fix}, and the source code versions V_{n-1} and V_n, which represent the source code versions of two consecutive revisions in the project's version control system. The difference between V_{bug} and V_{fix} is the isolated bug fix, which does not contain unrelated changes.

3. DATABASE OF REAL BUGS

For the initial version of Defects4J, we reproduced and isolated 357 bugs for 5 open source programs. This section provides details about those programs and the bug metadata included in the bug database of Defects4J.

3.1 Artifacts

Table 1 lists all programs and the numbers of corresponding real bugs that are available in Defects4J's bug database. This bug database provides the following artifacts and metadata for each bug:

- **Revisions in project's version control system**
  The bug database provides a mapping from bug id to the revision ids of which V_{fix} and V_{bug} were derived. The mapping to the faulty revision is merely maintained for reproducibility — V_{bug} is obtained by re-introducing the bug (i.e., applying the patch of the isolated bug to V_{fix}).
- **Patch of isolated bug**
  The bug database provides the patch that represents the isolated bug — that is, the diff between V_{fix} and V_{bug}. Basic statistics on patch size and a list of modified classes are also available.
- **Tests that expose the bug**
  The bug database provides a list of individual tests that expose the bug (a bug might be accompanied by more than one test that exposes the bug). For each test, the name, root cause, and stack trace is available.

Table 1: Programs and number of real bugs available in the initial version of Defects4J.

Program	Bugs	KLoc*	Test KLoc*	Tests	Dev years
JFreeChart	26	96	50	2,205	7
Closure Compiler	133	90	83	7,927	5
Commons Math	106	85	19	3,602	11
Joda-Time	27	28	53	4,130	11
Commons Lang	65	22	6	2,245	12
Total	357	321	211	20,109	

*KLoc for the most recent version, as reported by SLOCCount

3.2 Implementation Details

Defects4J's bug database does not use a particular version control system to store faulty and fixed program versions and accompanying tests. Rather, it references the projects' version control systems to ensure flexibility and extensibility. The projects' version control systems are heterogeneous and therefore abstracted by the database abstraction layer.

4. DATABASE ABSTRACTION LAYER

Section 3 described the artifacts that Defects4J's bug database provides. This section details the database abstraction layer to access those artifacts. The purpose of this abstraction is to provide hassle-free access to faulty (V_{bug}) and fixed (V_{fix}) program versions as well as uniform access to the program's build systems. Directly accessing the real bugs, and the faulty and fixed program versions would require considerable effort due to the heterogeneous version control and build systems of the programs.

4.1 Provided API

The abstraction layer of the bug database provides the following components:

Abstraction of version control systems
This component provides a uniform interface for checking out V_{bug} and V_{fix}, thus enabling the user to access V_{bug} and V_{fix} of any project in the database without having knowledge about the underlying version control system of that project.

Abstraction of build systems
This component abstracts the projects' build systems and provides uniform access to common build targets such as the compilation of sources and tests, or the execution of tests.

4.2 Implementation Details

In order to provide a uniform interface for accessing V_{bug} and V_{fix}, Defects4J assigns a unique id to each bug, abstracting over version control specific revision numbering schemes. Using the bug metadata, the database abstraction layer maps those unique bug ids to revision ids of the project's version control system, and also applies necessary patches to provide V_{bug} and V_{fix}.

The open source programs included in Defects4J employ different build systems, which vary between programs and even between revisions of the same program. For example, over the last 10 years, the build system of the Apache commons libraries switched from Ant to Maven (and back again). In addtion, the developers also restructured the source code directories and renamed packages. Due to the complexity and diversity of the build configurations, creating a general build file for all revisions and all projects is not practical. Therefore, Defects4J uses a hierarchy of build files. Defects4J's build file at the top of the hierarchy provides the uniform interface and the build file of the checked-out program version always represents the bottom. Intermediate build files abstract program-specific configurations. Creating such intermediate build files for a specific program requires manual effort — this is, however, a one-time effort.

5. TEST EXECUTION FRAMEWORK

Defects4J features a test execution framework that provides several components to perform common tasks in software testing research. The main goal of this framework is to reduce researchers' effort of (re-)implementing common tasks such as test generation, test execution, and code coverage or mutation analysis.

5.1 Provided API

The test execution framework currently provides the following components:

Monitoring test execution
Defects4J provides a component to monitor the execution of test suites or individual test cases. This component executes a set of tests and monitors the class loader. It returns detailed information about failing tests and also a list of program and test classes that were loaded during test execution. This component eases the determination of failing tests including the root cause and stack trace.

Test Suite Manipulation
Defects4J provides a component to manipulate and merge test suites — individual tests can be removed or replaced. A common use case for this component is to automatically remove all failing tests from a (generated) test suite.

Test Generation
Defects4J provides a component for test generation — tests can be generated for each faulty or fixed program version. Defects4J employs EvoSuite [3] as the default back-end for automated test generation.

Mutation Analysis
Defects4J provides a component for mutation analysis — mutation analysis can be performed for arbitrary test suites on each program version. Defects4J uses Major [5, 7] as the default mutation testing framework.

Code Coverage Analysis
Defects4J provides a component for measuring code coverage — code coverage metrics can be determined for arbitrary test suites on each program version. Defects4J supports Cobertura[2] and CodeCover[3] for the code coverage analysis.

5.2 Implementation Details

The test execution framework builds on top of the database abstraction layer (described in Section 4), provides utilities, and abstracts the use of external testing tools such as Cobertura, CodeCover, EvoSuite, or Major. The test execution framework provides a uniform interface to the external tools and their generated data, and each component performs all necessary steps such as instrumentation, test execution, and data analysis.

[2]`http://cobertura.sourceforge.net`, [3]`http://codecover.org`

6. RELATED WORK

The software-artifact infrastructure repository (SIR) [2] can be considered the first attempt to provide a database of real bugs to enable reproducibility in software testing research. SIR currently provides 81 subjects written in Java, C, C++, and C#, but most of the faults are hand-seeded or obtained from mutation. The number of real bugs for Java subjects is 35 and the median size of those subjects is 120 LOC, ranging between 24 and 8,570. Besides, none of the program versions that provide real bugs is accompanied by any tests, and SIR does not provide a uniform build system interface. In contrast to SIR, Defects4J provides 357 real bugs for 5 large real-world programs ranging between 22,000 and 96,000 LOC. Moreover, all 5 programs feature comprehensive test suites and each bug is reproducible with an exposing test case.

The iBugs project [1] is the closest related work for a database of real bugs for Java programs. The authors created iBugs to provide a benchmark for fault localization techniques. It contains 223 bugs with an exposing test case. Bugs are also extracted from version control history but not isolated. Moreover, the implementation for populating the iBugs database is not publicly available, and the faulty versions can only be built with an outdated version of the JVM. Compared to iBugs, Defects4J has a broader scope of application and the following three advantages. First, for 5 programs, which differ in size and operation purpose, Defects4J provides 357 bugs, all accompanied by comprehensive test suites of which at least one test case exposes the bug. Second, Defects4J accounts for the fact that developers do not always minimize their commits — all bugs in Defects4J are isolated (i.e., they do not include unrelated changes such as features or refactorings). Third, Defects4J provides a comprehensive test execution framework with several built-in components to support common tasks in software testing research.

The Siemens benchmark suite [4] is another set of faulty programs. It consists of 7 C programs, whose sizes vary between 141 and 512 LOC. However, the authors obtained faulty program versions by manually seeding faults, which they described as being very similar to simple mutations.

7. CONCLUSIONS AND FUTURE WORK

This paper presents Defects4J, a database and extensible framework to enable controlled testing studies for Java programs. The initial version of Defects4J contains 357 real bugs for 5 large open source programs, and its comprehensive framework allows an easy integration of those bugs in various software testing studies. The most important feature of Defects4J, in our view, is extensibility. Since Defects4J builds on top of the projects' version control and build systems, new bugs can be added with little or no effort.

Adding a new program to Defects4J requires a one-time manual effort for providing a program-specific wrapper build file but reproducing bugs is an automated step. The inclusion of a new bug into Defects4J's database for an already existing program is automated if the following requirements are fulfilled:

- The build system configuration does not change.
- The bug fix does not include features and refactorings.
- The fixed program version is accompanied by at least one test case that exposes the bug in the faulty version.

We assume that the build system configuration of a program does not frequently change. This assumption is supported by our experience with the 5 programs included in the initial version of Defects4J — the build system configuration changed at most 4 times over a development period of 12 years.

Committing minimized bug fixes together with a regression test that exposes the bug is already considered best practice for some of the included programs. Yet, Defects4J provides utilities to ease the process of bug isolation for the cases where a developer commits a bug fix along with unrelated changes.

Adding further programs to enable an increase of the number of bugs in Defects4J is part of our future work. Besides increasing the number of programs and bugs in Defects4J, improving the provided bug metadata by adding a classification [8] is another area for future work. Defects4J is publicly available on its website:

`http://defects4j.org`

8. REFERENCES

[1] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 433–436, 2007.

[2] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[3] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 416–419, 2011.

[4] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 191–200, 1994.

[5] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014. To appear.

[6] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? Technical Report UW-CSE-14-02-02, University of Washington, 2014.

[7] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–20, 2012.

[8] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1):41–84, 2005.